

Süsteemprogrammeerimine keeles C



11 Loeng

*Mis põhimõtteliselt sellest, ainult kõnelebki, mis on thread
ehk edaspidi "lõim"*

Eelmises osas (Protsesside juhtimine)

Protsessidele on võimalik saata tarkvaralisi katkestusi

Katkestust kirjeldab täisarv

Kui saabub katkestus, võib programm kas:

- seda ignoreerida
- jätta signaal haldamata, enamasti protsess lõpetatakse
- signaal kinni püüda: käivitada etteantud funktsioon

Eelmises osas: signal()

signal() abil on võimalik signaalidele *handler* määrata, signaali ignoreeritavaks märkida (SIG_IGN) või siis vaikimisi käitumise peale tagasi lülitada (SIG_DEFL)

argumentideks signaalinumber ja käivitatav funktsioon

käivitatav funktsioon saab argumendiks signaalinumbri

Eelmises osas: kill()

kill saadab etteantud protsessile või nende grupile signaali

positiivne pid: protsessinumber

pid 0: kõigile programmi alamprotsessidele

pid -1: kõik protsessid peale esimese (init)

negatiivne pid: protsessiga pid sama grupi protsessidele

saata saab juurkasutaja kõikidele, tavakasutaja enda omadele

Eelmises osas: alarm()

määrab kellaajalise signaali

määratakse mitme sekundi pärast saabub SIGALRM

0 sekundine intervall tühistab
tagastusväärtus aeg eelmisele alarmiajale

Eelmises osas (Long Jump)

setjmp() ja longjmp() paar võimaldab salvestada programmi *stacki* seisundi ning sellesse hiljem tagasi hüpata

setjmp() tagastab käivitamisel 0 ja pärast longjmp() toimumist viimase argumendis antud numברי

Eelmises osas (Make)

Make abil on võimalik suurte programmide
kompileerimine mugavaks muuta

Make jaoks kirjutatakse Makefile, kus on kirjeldatud
programmi failide omavahelised sõltuvused

käsurealt kompileerimiseks:
`make`

käivitatakse kõige esimene *rule*

Eelmises osas (Makefile)

Makefile sisaldab käskusid kujul:

target: prerequisite
command

käsu ees on *tab*

target ja prerequisite on enamasti failid, aga võivad olla ka lihtsalt käsud/eesmärgid

Eelmises osas (Eeldusreeglid)

kui eesmärgiks on midagi.o fail, kirjutame ainult header failide nimed.

midagi.c fail lisandub automaatselt nagu ka lihtsakoeline kompileerimiskäsk

eraldi kompileerimiskäsku ei ole sellisel juhul tarvis

Eelmises osas (.PHONY)

.PHONY on eesmärk, mille sõltuvused on kõik käsud
(st ei sõltu failide olemasolust)

.PHONY: clean ütleb, et clean ei ole fail, vaid
kompileerimise käsk/tegevus ja sellenimelise faili
olemasolul tuleb ikkagi tegevus täita

Eelmises osas (Makrod)

Makro on võimalus Makefile'is muutujaid
deklareerida

nimi = midagi

Kasutades asendub siia muutuja väärtus:
\$(nimi)

Eelmises osas (Make ja kataloogid)

Enamasti tehakse iga kataloogi (mooduli) tarbeks oma makefile, ning .h failid koondatakse ühte kataloogi kokku.

Programmi juurkataloogis on makefile, mis käivitab kogu kompileerimise. Kataloogide *makefile*id teevad vaid vajalikud tegevused antud kataloogi sisu kompileerimiseks

Eelmises osas (Autotools)

Automake ja Autoconf nimelised tööriistad teevad automaatselt Makefile'i ning konfiguratsiooniskripti

Programmeerija kirjutab configure.in ja Makefile.am failid

aclocal

autoconf

automake

./configure

make

make install

Tänases osas

- ♦ Lõimed

Tänases osas

- ♦ Lõimed on käivitatavad programmiosad, mis jooksevad ühe protsessi raames paralleelselt.
- ♦ Lõimel on oma lõime ID, *stack*, *stack pointer*, käsuloendur (*program counter*), seisundikoodid (*condition codes*) ja üldotstarbelised registrid.
- ♦ Ühe protsessi mitu lõime jooksevad protsessi kontekstis paralleelselt, jagavad sellega koodi, andmeid, *heapi*, jagatud teeke (*library*), signaali *handlereid*, ja avatud faile.

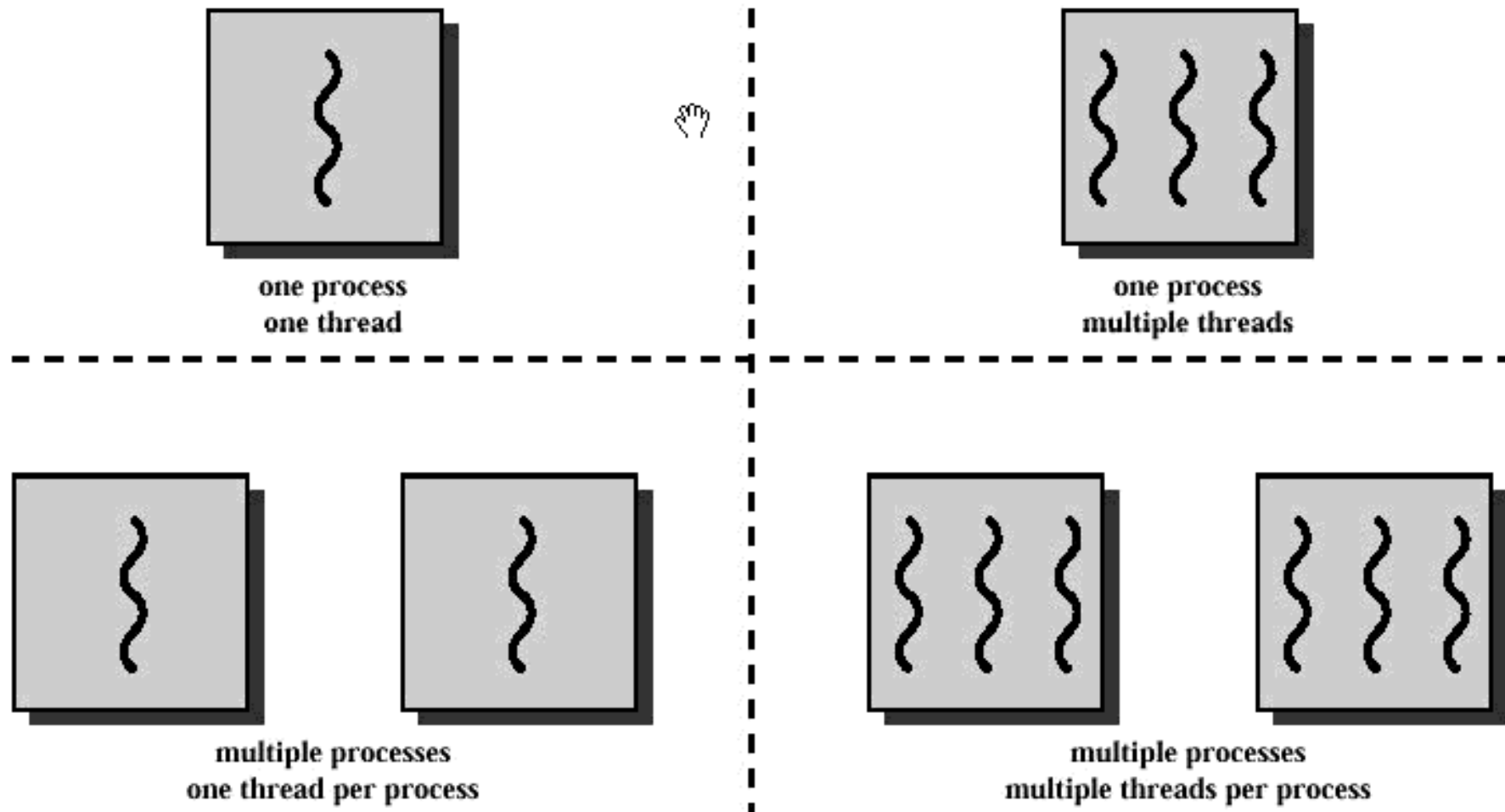
Protsess vs lõim

- ♦ Protsess – ressursside omanik
 - virtuaalne aadressiruum sisaldab protsessi *imaget*
 - ligipääs protsessori(te)le, teistele protsessidele, failidele ja I/O ressurssidele on kaitstud
- ♦ Lõim – protsessoriaja saaja
 - sisaldab käivitamise seisundit (jookseb, valmis, blokitud...)
 - salvestab lõime konteksti kui ei jookse
 - oma käivitus*stack* ja mõned lõimepõhised olekumuutujad
 - ligipääs protsessi mälule ja aadressruumile

Lõimede eelised protsesside ees

- ♦ Mõistliku implementeerimise puhul:
 - Võtab lõime loomine vähem aega kui protsesside oma, sest kasutatakse olemasolevat aadressruumi
 - Lõime lõpetamine võtab vähem aega
 - Kahe samas protsessis oleva lõime vaheline kontekstilülitus võtab vähem aega; osaliselt jagatud aadressiruumi tõttu
 - Vähem suhtlemisele kuluvat ressursi: lõimede vaheline suhtlus on väga lihtne, sest jagatakse pea kõike, eeskätt aadressruumi. Ühes lõimes tekkinud andmed on teise jaoks kohe kättesaadavad.

Single- vs Multithreading



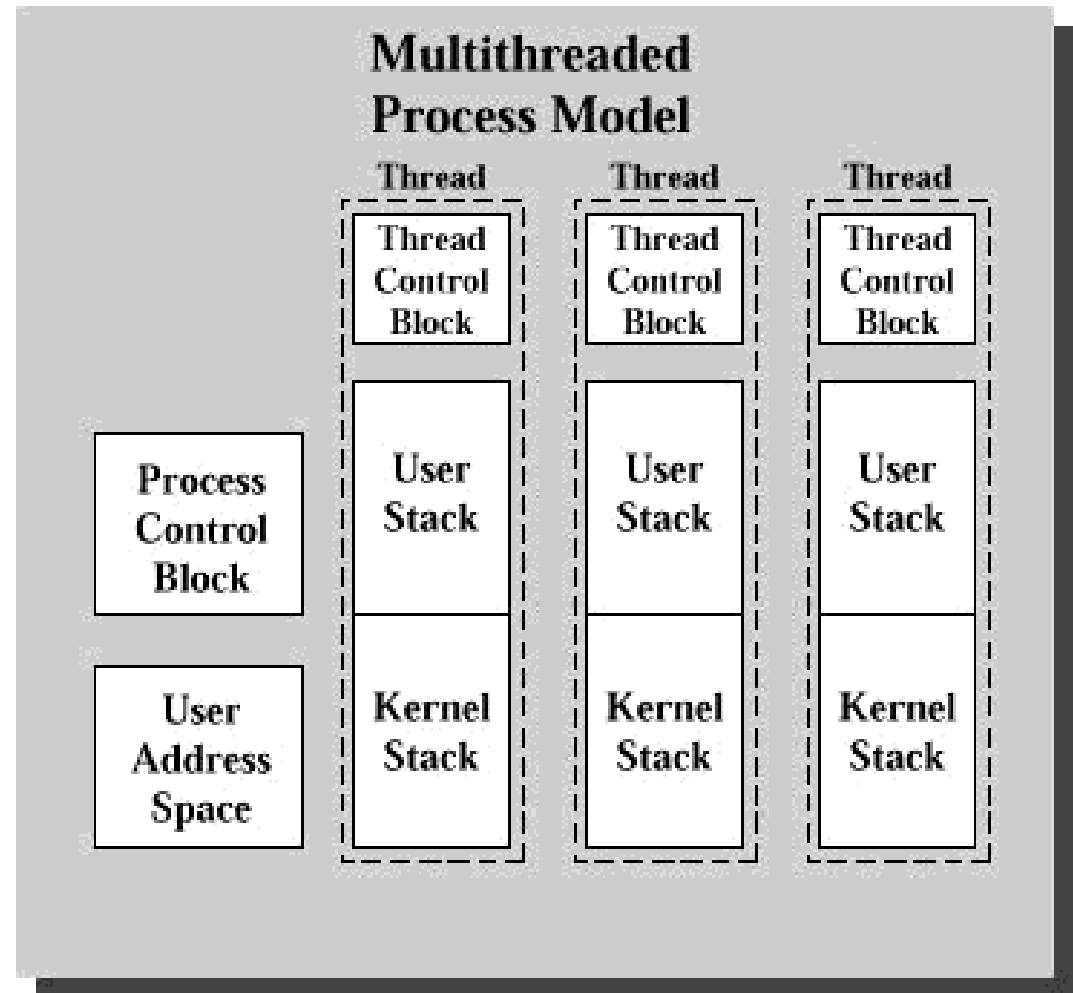
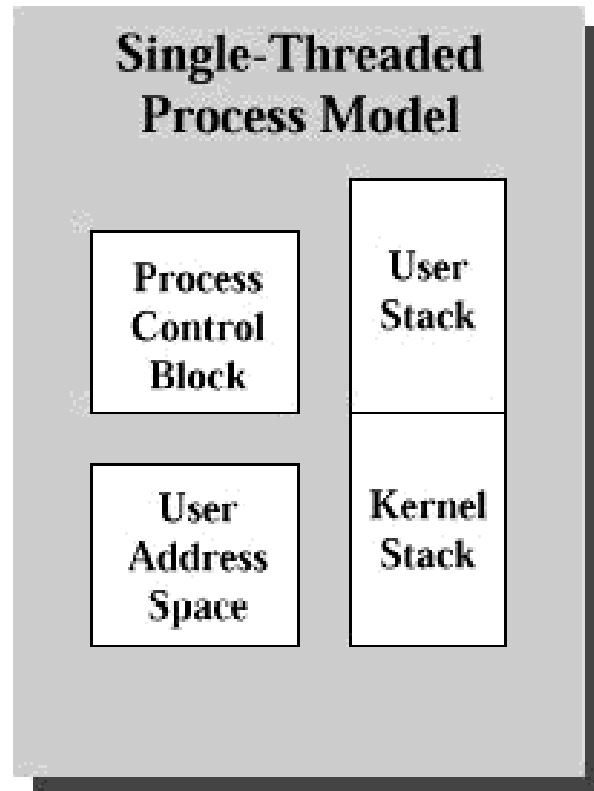
Multithreadingu eelised

- ♦ Programmi reageerimine paraneb
- ♦ Mitut protsessorit kasutatakse efektiivsemalt
- ♦ Programmistruktuur paraneb
- ♦ Kasutatakse vähem süsteemiressursse

Multithreadingu puudused

- ♦ Programmi ühe osa kokkujooksmine viib mälust kogu programmi
- ♦ Sünkroniseerimisprobleemid
- ♦ Turvalisusprobleemid

Single versus Multithreading



Lõimesid kasutav programm

- ♦ Näiteprogramm: LANi failiserver
 - Haldab lühikese aja jooksul saabuvaid mitmete failide nõudmisi. Iga soovi hallatakse eraldi lõimes.
 - Erinevad lõimed võivad joosta erinevatel protsessoritel.

Lõimede ...teegid

- ♦ Annavad lõimede liidese:
 - loovad ja hävitavad lõimesid
 - vahendavad lõimede vahel andmeid
 - ajastavad lõimede käivitamist
 - salvestavad ja taastavad lõimede kontekste
- ♦ Ei ole osa C standardist
- ♦ Näiteks:
 - POSIXi lõimed
 - SOLARISE lõimed

Lõimede juhtimine

- ♦ Pthreads (POSIX threads) defineerib ligikaudu 60 funktsiooni, mis võimaldavad C programmidel lõimi luua, hävitada, nende vahel kadudeta infot jagada ja kaaslastele süsteemi olekust teada anda
- ♦ Enamus lõimi kasutatavatest programmidest kasutavad sellest vaid väikest osa

Lõimede tegemine

```
#include <pthread.h>
int pthread_create(pthread_t * thread,
                  pthread_attr_t * attr,
                  void * (*start_routine)(void *),
                  void * arg);
```

- ♦ Loob uue lõime, tagastab määratud aadressil selle ID, käivitab ettenäidatud funktsiooni eraldi lõimena ja tagastub. arg võib olla ka NULL
- ♦ Tagastab 0 kui edukas ja midagi muud, kui viga

```
pthread_t pthread_self(void);
```

- ♦ Tagastab parajasti jooksva lõime ID

Lõimedega hello.c

```
#include <pthread.h>
#include <stdio.h>
void *thread(void *var gp);

int main() {
    pthread_t tid;

    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *var gp) {
    printf("Hello, world!\n");
    return NULL;
}
```

Lõimede seiskamine

- ♦ Lõim seiskub kui täitub üks järgmistest:
 - Kui tagastub kõige kõrgema taseme funktsioon
 - Kui mõni lõim kutsub välja *pthread_exit()* funktsiooni. Argumendiks tagastusväärtuse pointer. Kui põhilõim *pthread_exit()* välja kutsub, ootab protsess kuni kõik lõimed on lõpetanud ja annab protsessi tagastusväärtuseks tagastatava väärtuse.
 - Mõni kaaslõimedest kutsub välja kerneli *exit()* funktsiooni. kogu protsess lõpetab töö
 - Kaaslõim kutsub välja *pthread_cancel()* funktsiooni vastava lõime ID-ga.

Lõimede ... niitmine (reaping)

```
void pthread_exit(void *retval);  
int pthread_cancel(pthread_t thread);  
int pthread_join(pthread_t tid, void **thread_return);
```

- ♦ Teiste lõimede lõpetamist saab oodata käsuga `pthread_join()`
- ♦ Blokib, kuni etteantud thread lõpetab
- ♦ Erinevalt Unixi `wait()` käsust saab oodata ainult ettemääratud threadi järel
- ♦ Suvalise threadi järgi ootamine polegi võimalik

Lõimede lahtiühendamine (detach)

```
int pthread_detach(pthread_t tid);
```

- ♦ Lõim võib olla kas *joinable* või *detached*. *Joinable* tähendab, et teda saab killida ja niita. Tema mäluressurss (stack näiteks) on kinni kuni mõni teine lõim selle puhastab.
- ♦ *Detached* lõime samas niita ega killida ei saa. Mälulekete vältimiseks tuleb iga *joinable* thread puhastada või *pthread_detach()* funktsiooniga lahti ühendada.

```
pthread_detach(pthread_self()) // used to detach self
```

- ♦ Reeglina tahad kasutada *detached* lõimesid.

Jagatud muutujad

- ♦ Muutujate jagamine on põhiline lõimede eelis
- ♦ Samas võimaldab ta tekitada ohtralt raskesti avastatavaid vigu
- ♦ Jagatud on globaalsed muutujad
- ♦ Kohalikud automaatsed muutujad (*stack*) ei ole jagatud, aga ka mitte kaitstud (virtuaalne aadressiruum on ju jagatud)
- ♦ Kohalikud staatilised muutujad jagatud nagu globaalsed
- ♦ Üldreegel: muutuja on jagatud siis ja ainult siis, kui rohkem kui üks lõim temale viidet omab

Veaolukord jagamisel

```
#include <pthread.h>
#define NITERS 10000000
void *count(void *arg);
/* shared variable */
unsigned int cnt = 0;
int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    if (cnt != (unsigned) NITERS*2)
        printf("BOOM cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
}
void *count(void *arg) { // thread routine
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL; }
```

Veaolukord jagamisel

- ♦ Vastav kood:

```
for (i = 0; i < NITERS; i++)  
    ctr++;
```

- ♦ Tegelik kood:

```
LOAD ctr
```

```
INCREMENT ctr
```

```
STORE ctr
```


Mutexid

- ♦ Mutex on sünkroniseerimismuutuja, mida kasutatakse jagatud muutujate kaitsmisel. Mutexil on 3 lihtsat operatsiooni:
 - init
 - lock
 - unlock

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        pthread_mutexattr_t *attr);
```

- ♦ Kompileerimise ajal initsialiseerimine:

```
pthread_mutex_t mutex =  
    PTHREAD_MUTEX_INITIALIZER;
```

Mutexi lukustamine ja vabastamine

- ♦ Lukustamine:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- ♦ Vabastamine:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- ♦ Need on atomaarsed funktsioonid
- ♦ Lukustamine on tuntud ka kui *acquire* ja vabastamine kui *release*.
- ♦ Üks mutex saab korraga olla ainult ühe lõime käes

Mutexi kasutamine

```
// kuskil väljaspool lõime  
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex, NULL);  
  
// lõimes sees  
pthread_mutex_lock(&mutex);  
    // kriitiline segment  
    // jagatud muutujate töötlemine  
pthread_mutex_unlock(&mutex);
```

Parandatud näide

```
/* thread routine */  
void *count(void *arg)  
{  
    int i;  
  
    for (i=0; i<NITERS; i++) {  
        pthread_mutex_lock(&mutex);  
        cnt++;  
        pthread_mutex_unlock(&mutex);  
    }  
    return NULL;  
}
```

Condition variables

- ♦ *Condition variable* on muutuja, mida kasutatakse lõimede omavaheliseks sünkroniseerimiseks.

```
int pthread_cond_init(pthread_cond_t *cond,  
                      pthread_condattr_t *attr);  
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);  
int pthread_cond_signal(pthread_cond_t *cond);
```

Condition variable kasutamine

```
pthread_mutex_lock( &mutex );  
pthread_cond_wait( &cond, &mutex );  
pthread_mutex_unlock( &mutex );
```

- ♦ `cond_wait` laseb mutexi lukust lahti JA blokeerib lõime
- ♦ lõim ootab tingimusmuutuja *cond* muutumist.
- ♦ hiljem saadab mõni teine lõim tingimusmuutuja *cond* järgi ootavale lõimele signaali ülesärkamiseks

```
pthread_cond_signal( &cond );
```

- ♦ äratatakse täpselt 1 lõim, millele antakse mutex tagasi.

```
pthread_cond_broadcast( &cond );
```

Meeldivat õhtu jätku